INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Realistic Implementations of External Memory Algorithms using a Coarse Grained Paradigm

Jens Gustedt

No 4719

Février 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4719--FR+ENG

Towards Realistic Implementations of External Memory Algorithms using a Coarse Grained Paradigm

Jens Gustedt*

Thème 1 — Réseaux et systèmes
Équipe AlGorille

Rapport de recherche n°4719 — Février 2003 — 11 pages

Abstract: We present an extension to SSCRAP, our C++ environment for the development of coarse grained algorithms, that allows for easy execution of programs in an external memory setting. Our environment is well suited for regular as well as irregular problems and scales from low end PCs to high end clusters and mainframe technology. It allows running algorithms designed on a high level of abstraction in one of the known coarse grained parallel models without modification in an external memory setting. The first tests presented here in this paper show a very efficient behavior in the context of out-of-core computation (mapping memory to disk files), and even some (marginal) speed up when used to reduced cache misses for in-core computation.

Key-words: Coarse grained parallel models, external memory algorithms, experiments.

(Résumé : tsvp)

* INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr – Phone: +33 3 83 59 30 90

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : 03 87 20 35 00 - International: +33 3 87 20 35 00
Télécopie : 03 87 76 39 77 - International : +33 3 87 76 39 77

Vers une implantation réaliste d'algorithmes à mémoire externe utilisant un paradigme à gros grain

Résumé : Nous présentons une extension de SSCRAP, notre environement C++ pour le développement d'algorithmes à gros grain, qui permet une execution facile de programmes dans le cadre d'utilisation de mémoire externe. Notre environnement est bien adapté à des problèmes réguliers et irréguliers et fonctionne aussi bien au PC bas de gamme que sur des grappes haut de gamme et des super-calculateurs. Sans aucune modification du code, il permet de faire tourner des algorithmes qui ont étés conçus pour un des modèles parallèles à gros grain avec de la mémoire externe. Les premiers tests présentés ici montrent un comportement très efficace dans le contexte du calcul « out of core » (affecter de la mémoire à des fichiers sur disque dur) et ils montrent même une accélération (marginale) quand à l'utilisation « in-core » pour éviter des défaut de caches.

Mots-clé : modèles parallèles à gros grain, algorithmes à mémoire externe, expérimentation

1 Overview

In the mid-nineties several authors, see Cormen and Goodrich [1996], Dehne et al. [1997], developed a connection between two different types of models of computation : BSP-like models of parallel computation and IO efficient external memory algorithms. Their main idea is to force data locality during the execution of a program by simulating a parallel computation of several processors on one single processor.

Whereas such an approach is convincing on a theoretical level, its efficient and competitive implementation is quite challenging in practice. In particular, it needs software that induces as less computational overhead as possible by itself. Up to now, it seems that this has only been provided by software specialized in IO efficient implementations.

Currently, with our library SSCRAP, see Essaïdi et al. [2002], we reached a level of scalability that let us hope that it could now be possible to attain high efficiency in both (or even mixed) contexts :

- SSCRAP can run hundreds of "processors" (as POSIX threads) on a single machine (main-frame or not) without loosing on its performance,
- It can handle problem instances efficiently that may exceed the size of the address space of an individual hardware processor.

In fact, with some relatively small add-ons to SSCRAP we were able to provide such a framework. It was tested successfully on some typical hardware, PC with some GB of free disk.

The main add-on that was integrated into SSCRAP was a consequent implementation of an abstraction between the *data* of a process execution and the memory of a processor. The programmer acts upon these on two different levels :

- with a sort of *handle* on some data array which is an abstract object that is common to all SSCRAP processors.
- with a map of its (local) part of that data into the address space of the SSCRAP processor, accessible as a conventional pointer.

Another add-on was the possibility to fix a maximal number of processors (ie threads) that should be executed concurrently. With these add-ons, simple environment variables `SSCRAP_MAP_MEMORY` and `SSCRAP_SERIALIZE` allow for a runtime control of the program behavior.

In the present paper we present promising test with three different problems/algorithms :

sorting, for a problem that has a non-linear lower bound in terms of computation, but allows for only linear amount of streaming IO when doing reasonable assumptions on the granularity on the setting.

permutation generation at random, for a problem that is linear, has a highly random memory access pattern, but where the needs of computation (random numbers) and memory access equilibrate to a certain extent.

list ranking, for a problem that is linear, has a highly random memory access pattern and where this memory access pattern completely dominates execution.

2 Introduction

The main purpose of SSCRAP, see Essaïdi et al. [2002], is to provide a development framework for the various so-called coarse grain models and to guarantee the portability and efficiency of the produced programs on a large variety of parallel and distributed platforms. It is not the first library that provides such a framework : several libraries implementing communications for coarse grained models were developed, see Miller [1993], Goudrau et al. [1995], Hill et al. [1997], Bonorden et al. [1999].

When implementing coarse grained algorithms, we noticed that communication rounds often have the same flavor : the most frequently performed data exchange is bulk all-to-all. During this kind of exchange, the processors generally don't have any preliminary (static) knowledge of the size of the data to be sent or received.

Starting from this and other observations, a parallel programming model called PRO (Parallel Resource-Optimal computation, see Gebremedhin et al. [2002]) was proposed and developed in close interaction with SSCRAP. PRO allows the design and analysis of scalable and resource-optimal algorithms. It is based on

– relative optimality compared to a fixed sequential algorithm as an integral part of the model ;
– measuring the quality of the parallel algorithm according to its granularity, that is the relation between p and N.

PRO only considers algorithms that are both time- and space-optimal. As a consequence of this enforced optimality, a PRO-algorithm always yields linear speed-up relative to a reference sequential algorithm. In addition, PRO assumes that the coarse grained communication cost only depends on p and the bandwidth of the considered interconnection network. In fact, the experiments based on SSCRAP confirm that under these circumstances the communication start-up and latency can both be neglected from the cost analysis.

Thus PRO algorithms with some easy-to-follow *golden rules* and their implementation as SSCRAP programs ensure the efficient use of nowadays parallel and distributed hardware. It is easy to see, that the same observations as for parallel or distributed executions also apply to the cost analysis of simulations. A simulation of a multi-processor algorithm on a machine with a small number of real processors (or just one) behaves almost the same in terms of work load of the machine as a whole.

The additional cost (again under reasonable assumptions) of a simulation is caused by the overhead for the maintenance of the memory of the virtual processor. In an external memory context, a simulation of a BSP or PRO superstep is equivalent of reading the memory for such a processor from disk, simulating the superstep itself as performed by that processor and then writing its modified memory back on disk. So the overhead for each superstep are streaming read and write operations, the most efficient operations concerning file-IO.

The overall cost of these operations is the sum of the sizes of all memory needed by processors in any of the supersteps. If this sum is of the same order of magnitude as the known sequential cost of the algorithm, the simulation is efficiently using streaming IO in external memory. Such a requirement is fulfilled by two important classes of the known coarse grained parallel algorithms :

Algorithms with constant number of supersteps. Clearly for such algorithms the overhead is bound by the number of supersteps. In our test set two algorithms fall into that category : for *sorting* we use ideas of Gerbessiotis and Valiant [1994] and for *randomized permutations* we use Gustedt [2002].

Algorithms with a recursive data reduction. These are algorithms that use recursion on a problem of reduced size to achieve their goal. If the data size in the recursion is some ε-factor smaller, $\varepsilon < 1$, then usually a bound with a geometric series proves a constant bound on the overhead. A good implementation of such an algorithm in an external memory setting is more challenging : the constant in the overhead depends on ε and we have to ensure that the input of reduced size need for the recursion is packed consecutively in memory. From our test set the *list ranking* algorithm falls into that setting. We use the implementation described in Guérin Lassous and Gustedt [2002].

3 Implementation

For the SSCRAP design, the main goals were portability, efficiency and scalability. The most difficult task was to balance between efficiency and portability. Indeed, to be portable, a program must consider a generic machine model disregarding the specificity of the physical architecture. However to be efficient, a program must get the full benefit from the available resources which differ enormously from one architecture to another. SSCRAP ensures this difficult task by introducing an abstract communication layer between the “user” routines of SSCRAP and the target platform. Currently, SSCRAP is interfaced with the two main types of parallel architectures : distributed memory (as e.g. cluster of PC or workstations) and shared memory.

For the context of this study only the shared memory interface is of relevance. That interface is build upon *PosiX threads* which ensures portability and efficiency on most modern platforms. In fact, usually SSCRAP can run with some hundred SSCRAP processors (ie threads in that case) without significant performance degeneration on any of the platforms that are supported : ranging from mono-processor PC or workstations, over few-processor machines like quadri-processor PCs to multiprocessor mainframes.

3.1 Serializing execution

The first thing to add to the shared memory interface was a tool to allow for the sequential "in-order" execution of the SSCRAP processors. Since each processor to be simulated corresponds to a thread, this was relatively simple to achieve with a semaphore. Whenever a SSCRAP processor enters one of the functions that define the end of a superstep it `post`s (increases) that semaphore, whenever it starts a new superstep it `wait`s for the same semaphore. At program startup the semaphore is initialized by the value κ found in environment variable `SSCRAP_SERIALIZE` (or infinity if doesn't exist).

As a consequence, at most κ SSCRAP processors will execute concurrently inside a superstep at any time. Care is taken such that this statement holds for any of the time consuming operations, especially interprocessor copying of data. We can simply think of the SSCRAP processors passing κ execution tokens to each other. Each processor keeps its token as long as it can during a superstep. It only passes it to the next processor when it is obliged to wait that its communications are terminated. Then it waits for its next turn for a token and enters the next superstep.

On the other hand, in the very brief period in between the supersteps when SSCRAP handles its control datastructures all SSCRAP processors are active and no deadlock occurs.

3.2 Communication Abstraction

Probably the most important feature of SSCRAP in the present context is the abstraction that it enforces from the underlying communication. In each superstep, the program may perform any sort of local computation for a given SSCRAP processor. In addition each such processor may issue one `send` and one `receive` operation to and from any processor. So in each superstep each processor issues at most p `sends` and `receives`, for p the amount of SSCRAP processors.

These two types of operations are non-blocking and unbuffered such that the processor can continue its operation until the end of the superstep. On the other hand it may not rely on the contents of the underlying send or receive buffers during the same superstep. In that way, the specific communication library can handle communications efficiently according to the specific requirements of the setting, without making unnessary copies of the data in buffers or blocking the processor while it might have useful work to perform.

In an external memory context this means that write or read operations (here `send` and `receive`) are scheduled long before the data will effectively be used by the program in the next superstep.

3.3 Data Abstraction

The main add-on that was integrated into SSCRAP was a consequent implementation of an abstraction between the *data* of a process execution and the memory of a processor. The programmer acts upon these on two different levels :

`sscrap : :array` gives a kind of *handle* on some data array which is common to all SSCRAP processors.

`sscrap : :apointer` maps its (local) part of that data into the address space of the SSCRAP processor and lets the programmer access this part like she would with a conventional pointer.

Such a two level model is in contrast to the usual "C" approach where arrays and pointers are (almost) the same type of object.

```
void identity (sscrap::array< long > &A){
  sscrap::apointer< long > ap(A);
  for (pos_type i=0; i<A.localSize(); ++i) ap[i]=i;
}
```

TAB. 1 – Programming example

It has the big advantage to free the run-time library from certain obligations. E.g two different `apointers`, `ap` and `bp`, that are constructed from the same `array` object at different points of the program may see the same data *mapped* to completely different addresses in the address space of the processor. In that way the system may re-use addresses and physical memory according to the needs of the execution.

3.4 Design

To guarantee portability and efficiency SSCRAP is written in C++. This allows us to have a design organized in different levels of abstraction, from the interface to the communication library (threads, MPI or PM^2) over classes organizing the supersteps to user interfaces such as `sscrap : :array` as described above. On the other hand, care is taken that modern C++ compilers are able to transform this code into efficient assembler.

For the use of external memory a demand for a pointer (address) to the local part of an array can be translated into a file mapping request by means of the system call `mmap`. This is done without changing the code or even recompiling it. Upon *creation* of a `sscrap : :array` the environment variable `SSCRAP_MAP_MEMORY` is inspected. If it is set, the memory for the array is allocated in a disk file and otherwise on the heap (with `malloc`). Since such events of creation are quite rare, the fact of deciding upon the memory model at runtime is not noticeable.

The C++ design of these classes hopefully guarantees that these constructs are easy to use. Consider the function definition in Table 1 as an example. It ensures that first the local part of `A` is mapped into memory if necessary, that it can be used efficiently inside the `for`-loop, and that it is unmapped at the end. In general, compilers are able to optimize such a code as given here such that all overhead (calls to `mmap`, `munmap` etc) is done outside the loop and such that all information (here `i`, a pointer and the size of the `array`) are hold in registers.

4 Experiments and analysis of the result

The implementation as presented here runs successfully on different types of platforms. We tested it on PC (mono- and biprocessor), multiprocessor workstations (SUN) and a mainframe with 56 processors (SGI). But since the tests are quite time consuming (one run several hours) and quite sensible to perturbations we had to restrict to a machine that could be dedicated during a whole week to these tests.

The experiments as presented were taken out on a conventional modern PC with the following characteristics :

CPU	Pentium III, 1.7 GHz
RAM	512 MB
disk	1 GB (swap), 1.6 GB available file system, 40 MB/sec
OS	GNU/linux v. 2.4.
C++	gcc v. 2.96

With these characteristics reading the whole contents of the RAM from disk needs at least 12.8 seconds and a total exchange of RAM with disk 25.6 seconds. The maximal disk throughput is 10.4 mio `long` or 5.2 mio `double` per second, or equivalently a streaming IO operation has a cost of 162 clock cycles per `long` and 325 per `double`.

Since the machine had no other charge, variances between different runs of the same problem instances were low. Therefore, with only some exceptions, experiments had only to be taken out 5 to 10 times. Exceptions were cases for which the running times were several hours per experiment, here we reduced the number to 3.

The results presented here are simply the wall clock time that the experiments did take. They were scaled by the number of items in the experiment, to obtain a cost factor of seconds per item. Curves are presented with doubly-logarithmic scales to better cover the different orders of magnitude of the experiments.

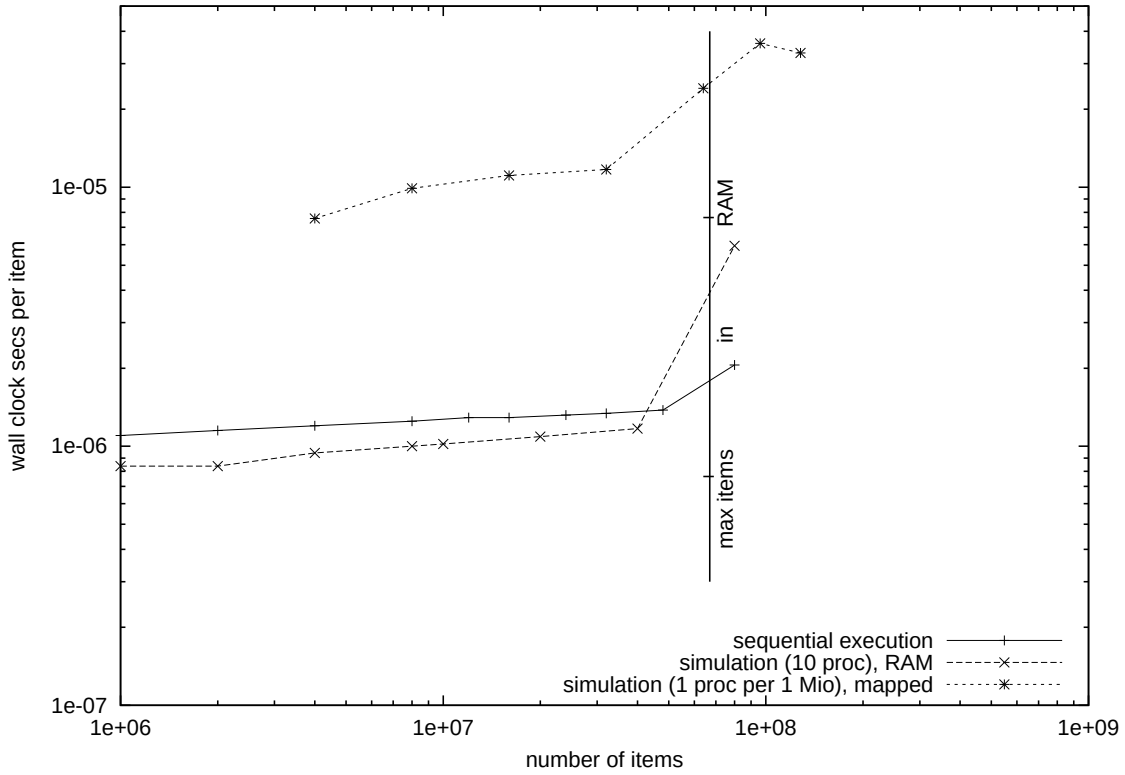

FIG. 1 – Sorting

All parallel algorithms have the common property that they use much more memory than the sequential ones. Generally spoken they are not *in place* algorithms like the sequential ones, so they use at least twice as much memory. The maximal amount of items that could be placed in RAM is indicated by a vertical bar in each of the figures. In general, it was not possible to drive the in-core execution of the programs beyond that barrier. Figure 2 shows one example were we tried to reach as far as possible ; it was necessary to reboot the machine several times in doing so.

4.1 Sorting

Besides being very useful in practice, sorting is a very interesting problem from a conceptional point of view. It has a non-linear lower bound in terms of computation, but this bound doesn't apply to memory access or communication operations.

In fact the implementation of the algorithm of Gerbessiotis and Valiant [1994] that we were using for the experiments takes advantage of this subtle difference. It has an overall computation time that is dominated by the time of locally sorting on each processor. Thus in general this time is $O(N/p \log(N/p))$, N the number of items, p the number of processors.

On the other hand it allows for only linear streaming IO when doing reasonable assumptions on the granularity on the setting, in our case $p < \sqrt{N}$. So in fact the most expensive operation, memory access can be avoided and substantially reduced.

We performed sorting of a vector of `doubles`. The sequential sorting routine that is uses as a subroutine is an efficient in place *quicksort*. The maximal data size the machine could hold in memory is of 67 mio items. Figure 1 shows the results of the experiments.

First, we observe that the information theoretic lower bound of $\log(N)$ per item for the sequential algorithm and of $\log(N/p)$ for the parallel one is not of much relevance for the practical resolution of the problem. Second,

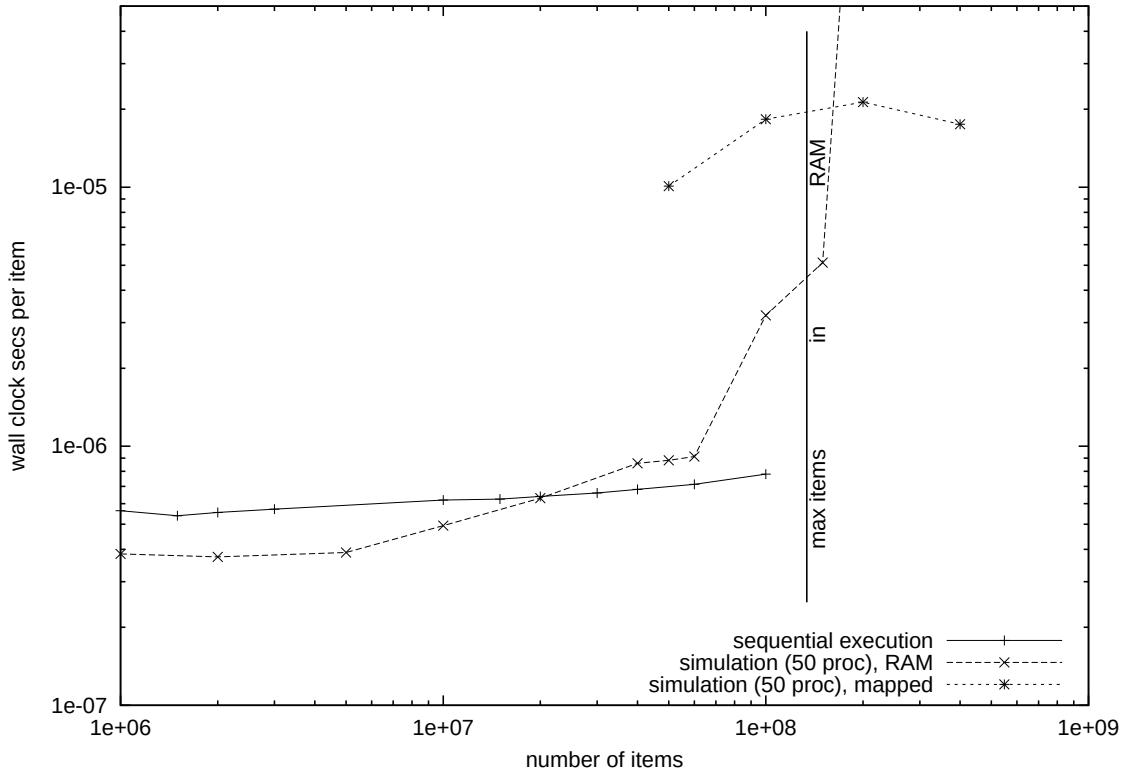

FIG. 2 – Randomized Generation of Permutations

we see that the simulation is already more efficient when handling problems that can be hold inside RAM. As a consequence, we conclude that our sequential algorithm is not the fastest possible ; some variant of multiway sorting (to which the simulation resembles) should be a bit more efficient.

The file mapping takes much more time than running the program entirely in RAM. Because of the average cost of 650 clock cycles for reading one item from disk this is not very surprising. On the other hand the corresponding running times were reliable far beyond the swap boundary, experiments only did stop when there was not enough disk space available.

4.2 Randomized Permutations

Randomized generation of permutations is a problem with a linear time complexity but where the most costly operation is random memory access. At least the standard sequential algorithm has a highly random memory access pattern and so in general most of these memory accesses will be cache or page misses.

But the needs of pure computation time for such a generation are also quite high, since we need (pseudo) random numbers. So memory access and computation should equilibrate to a certain extent.

For the tests we generated randomized permutations of `long int`s. The random number generation was done with linear congruent generators, one for each SSCRAP processor. The maximal data size the machine could hold in memory is of 134 mio items.

As for sorting, we see that even for in core computation the simulation is more efficient. The situation changes at a point were the simulation runs out of memory. Again, the mapping memory to files is a good strategy to handle problem sizes that otherwise could not be treated by the machine.

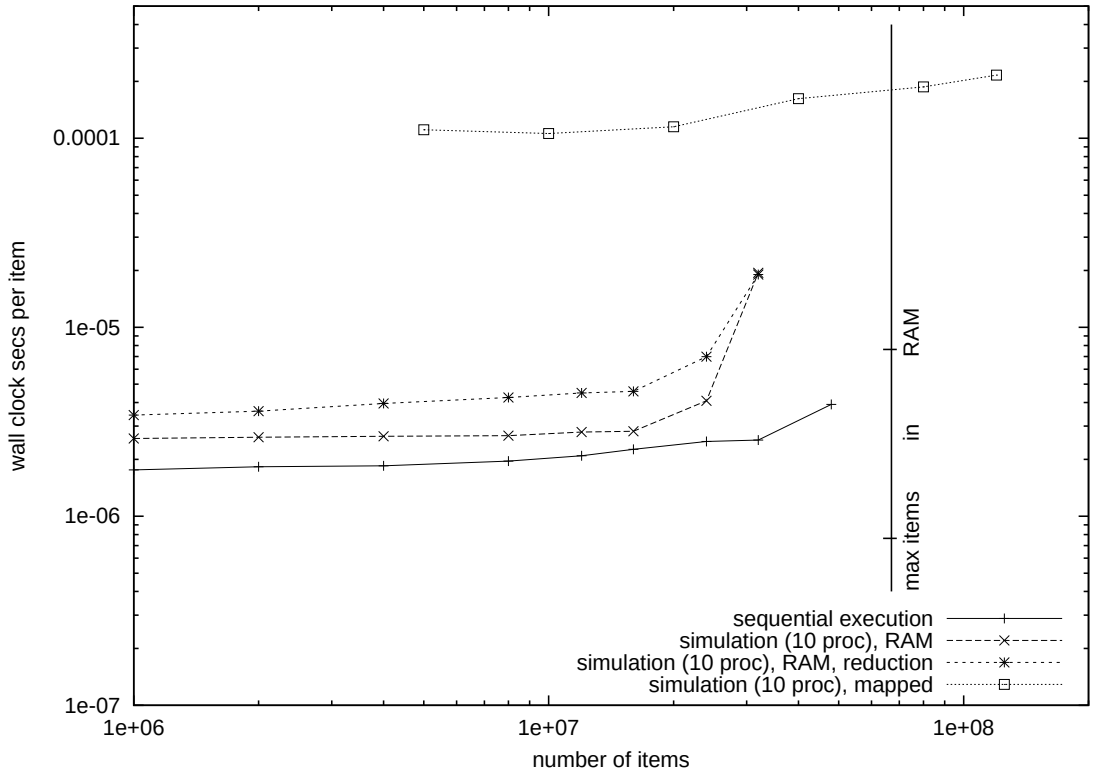

FIG. 3 – List Ranking

4.3 List Ranking

Given a collection of lists, the list ranking problem is to compute the distance to the tail of its component for every single item in these lists. This problem has a simple linear time solution, but since we may not assume any particular organization of the list items in memory, it has a highly random memory access pattern.

Even worse, since the principal operation here is the memory access itself, there are not much other operations that can be performed during a memory request. So the memory access pattern completely dominates execution time.

Since we need two `long` values to handle each item (one for the next in the list, and one for the distance), the maximal data size the machine could hold in memory is of 67 mio items.

List Ranking was the most demanding among the three problems. It has already a sequential running time which is quite hight and the implementation of an efficient parallel algorithm has not been simple at all. This complexity of the problem (or its solution) makes that the in-core simulation didn't bring us a gain. But all, the different in-core and out-of-core simulations showed good predictive behavior. In particular, mapping again enables us to handle problems that were not possible without.

5 Conclusion and Outlook

We have seen that SSCRAP enables us to extent the use of certain types of parallel programs to the setting of out-of-core computation. The principal bound in problem size that we encountered was related to the availability of a resource that is easily extensible and cheap : disk space.

In future work, we will try to explore more possibilities of this approach. First of all we will test it on hardware that has more potential concerning disk storage : larger and faster disks, hardware and software

RAID arrays. Having seen that our test machine was easily accessible during all the tests, we expect this to run without any mayor problems.

We are working on a new communication interface for SSCRAP that is based upon PM^2, see Denneulin and Namyst [1995]. This will allow to mix simulation and real parallelization in a PC cluster. Thereby we hope to be able to handle problems that will be larger than the address space (4 GB) of one individual (i86) processor : a cluster of k PC each having 4 GB of free disk could hold and sort a table of $2.6 \cdot 10^7 k$ `doubles`, which for $k > 4$ is more than the address space. This would be particularly simple to undertake for sorting since it only accesses the data items in a table. For the other algorithms the address of an individual item has to be encoded. If we continue using `unsigned longs` for that, the problems size is bounded by $2^{32} \approx 4.2 \cdot 10^9$ items. Moving to `long longs` which most compilers (and architectures) implement nowadays this bound can be extended to $1.8 \cdot 10^{19}$.

Another direction to follow will be an optimized use of the existing code. The choice of the inter-relation of the number of processors p that is simulated and the input size N could be important and should thus be investigated further.

Références

Olaf Bonorden et al. The Paderborn University BSP (PUB) Library—Design, Implementation and Performance. In *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, 1999.

Thomas H. Cormen and Michael T. Goodrich. A bridging model for parallel computation, communication, and I/O. *ACM Computing Surveys*, 28A(4), 1996.

F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarsegrained parallel algorithms. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3) :379–400, 1996.

Yves Denneulin and Raymond Namyst. PM^2 : Parallel multithreaded machine, un support d'exécution pour applications irrégulières. In *RenPar 7*, 1995.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP : An environment for coarse grained algorithms. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 2002.

Assefaw Hadish Gebremedhin, Isabelle Guérin Lassous, Jens Gustedt, and Jan Arne Telle. PRO : a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 106–113. IEEE, The Institute of Electrical and Electronics Engineers, 2002.

Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2) :251–267, 1994.

M. W. Goudrau, K. Lang, S. B. Rao, and T. Tsantilas. The green BSP library. Technical Report TR-95-11, University of Central Florida, Orlando, 1995.

Isabelle Guérin Lassous and Jens Gustedt. Portable list ranking: an experimental study. *ACM Journal of Experimental Algorithmics*, 7(7), 2002. `http://www.jea.acm.org/2002/GuerinRanking/`.

Jens Gustedt. Randomized permutations in a coarse grained parallel environment. Technical Report RR-4639, INRIA, November 2002.

Jonathan M. D. Hill et al. BSPlib : The BSP programming library. Technical report, Oxford University Computing Laboratory, 1997. URL `http://www.bsp-worldwide.org/standard/bsplib_C_examples.ps.Z`.

Richard Miller. A library for bulk-synchronous parallel programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, 1993. URL `http://www.comlab.ox.ac.uk/oucl/oxpara/ppsg.ps.gz`.

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103–111, 1990.

Unit´e de recherche INRIA Lorraine, Technopˆole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhˆone-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399